by
Luisa Simone

Lab Notes

A Guide to PostScript For Non-PostScript Programmers

■ If you understand the basics of its Page Description Language, you can troubleshoot and modify PostScript code with only your word processor.

To state it bluntly, PostScript has become the de facto standard of the electronic publishing industry.

There are still alternatives, of course. Before the advent of Adobe PostScript, the electronic publishing industry got along very well, thank you, using minicomputers and proprietary software with professional-level systems like Atex and Scitex to generate high-resolution documents on Linotronic and AGFA Compugraphic equipment. These systems won't go away overnight. But in the long run they cannot compete with the broadening user base and increasing technical power of PC-based systems.

PostScript revolutionized the industry by making it possible to create many of the same documents that the professional systems produced, at the same quality level, on the same high-resolution output devices, using an off-the-shelf, low-cost personal computer.

While it generated a lot of excitement, however, PostScript's 1985 debut as an integral part of the Apple LaserWriter created a fair amount of confusion. Many people assumed that PostScript was part and parcel of a Macintosh system. Steven Jobs may have thought of the PostScript-driven Apple LaserWriter as a Mac-only peripheral. But John Warnock and Charles Geschke—Adobe's founders and PostScript's creators—had a far more encompassing vision, and it was only a short time before IBM PCs were driving PostScript-capable laser printers.

PostScript's ability to migrate easily from the Mac to the PC—or indeed to any other platform—is explained by one fact: PostScript is inherently a device-independent Page Description Language (PDL). While most PC programs send a finished bitmapped image directly to the printer, what PostScript-capable applications do instead is to generate an intermediate set of instructions, or code. The PostScript interpreter, stored in the printer's ROM, translates the routines into the final, necessary, printable bitmap.

Bitmapped images consist of a fixed number of dots arranged in a series or matrix. In the case of monochrome printer files, black dots can be either *on,* which means they will print, or *off,* which means the white of the paper will show. However, the number of dots never changes. When you output a 6-inch-square, 300-dot-per-inch image on a 600-dpi printer, the file's fixed resolution cannot take advantage of the increased capabilities of the output device. If you do print at the higher resolution, the image will be only 3 inches square, because twice as many dots can fit into a square inch.

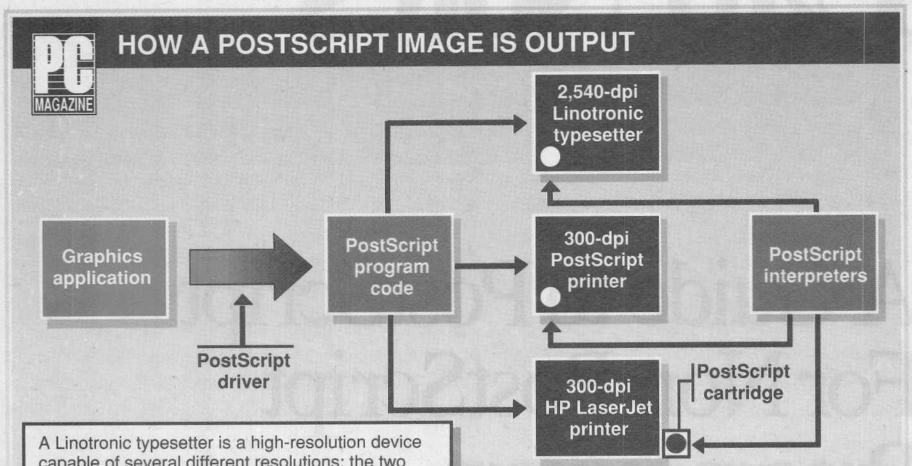
PostScript files, however, are vector files; they contain a list of instructions that an output device can interpret to produce an image. Since a description of a line, a square, or a circle has no intrinsic resolution, PostScript files are device- and resolution-independent.

As you can see in Figure 1, a PostScript driver, which is part of the application program (or part of the graphical operating system in the case of *Microsoft Windows*), actually writes only code. The PostScript file you generate from *PageMaker, CorelDRAW,* or *Micrografx Designer* is not a bitmapped representation of anything; it is uncompiled program code. The bitmap itself is created locally by the printing device.

The beauty of this system is that the intermediate code is independent both of the host system that created it and the ultimate output device. Indeed, though PostScript contains elements common to all languages (variables, loops, conditionals, and operators), and though many of its commands are graphics-specific (`moveto`, `lineto`, `arc`, and `scale`), there is rarely an instruction in a PostScript file that pertains to resolution. That's because the PostScript interpreter in the printer (which is also called a *RIP,* for Raster Image Processor) always assumes that it should output the vector instructions at the highest-possible resolution. It is this important aspect of the PostScript language that allows users to print out the same file on a 300-dpi laser printer or on a 2,540-dpi Linotronic typesetter.

It's instructive to contrast this approach with Hewlett-Packard's Printer Control Language (PCL). PCL has evolved in stages from a very simple language for line printers into its current (fifth) version, which contains sophisticated vector-based commands for imaging a page. The PCL 5 code itself can nevertheless be imaged directly *only* at 300 dpi—not coincidentally, the resolution of the HP LaserJet family of printers.

(In fairness, there is a workaround for printing out a PCL file at higher resolutions. The scheme involves scaling the im-



A Linotronic typesetter is a high-resolution device capable of several different resolutions; the two most common in the commercial printing field are 1,270 and 2,540 dots per inch. Linotype (the maker of Linotronic typesetters) has a PostScript interpreter or RIP (raster image processor) that converts PostScript code into the highest-resolution image the typesetter is capable of.

A typical PostScript printer, such as the Apple LaserWriter, is capable of only 300 dpi. For the purposes of this article, that's the only difference between a PostScript laser printer and a high-resolution typesetting device.

The cartridge currently in use with HP printers contains an interpreter. It can either translate from PostScript to HP's Printer Control Language (which then is imaged normally as though it were an HP file), or it can take over the printer engine, sending a fully processed bitmap file to the engine.

Figure 1: The secret of PostScript's flexibility is that applications generate an editable intermediate code that is turned into a final bitmap by an interpreter located in the printing device.

age at twice or three times the normal size and then printing out at a resolution that is a multiple of 300. A twice-normal-size image can be printed out on a 600-dpi printer, a four times normal size image can be printed out at 1,200 dpi. It isn't elegant, but it does work.)

Thanks to new charting, drawing, and DTP programs, chances are that you're already working with PostScript files in some form. You don't need to learn how to read a single line of PostScript code in order to do so. However, given the increasing market presence of PostScript clones, the varying quality of PostScript drivers, and Adobe's own June 1990 announcement of a new version of the language (see the sidebar "PostScript, Level 2"), learning about PostScript may be of more than academic interest. It may be a matter of old-fashioned job security.

Adobe's official list of OEMs and licensees is extensive. It includes front-line companies like IBM, Apple, NEC, Ricoh, and Fujitsu. There are, however, laser printers on the market (for example, the $7,995 Tektronix Phaser PX) that use a cloned version of Adobe PostScript instead of a licensed one. And clone interpreters come in all forms and at all prices.

You can add PostScript functionality to your tried-and-true HP LaserJet by purchasing expansion cards, cartridges, or software-only clone interpreters. Representative examples are the $2,500 Princeton Publishing Labs PS-388, the $495 PacificPage from Pacific Data Products, and the $445 QMS UltraScript PC Plus.

In large measure, the very success of the clone products is proof of PostScript's robust nature. However, their presence does increase the complexity of life in the PostScript age. Though it might be technically incorrect to label differences between Adobe PostScript and its imitators as incompatibilities (since they are deliberate changes), some clones substitute segmented curves for beziers, some choke on certain pattern-fill commands, and most can't handle true Adobe Type 1 fonts.

A much more basic issue is raised by the fact that in the five years since PostScript was introduced, there have been several extensions to the language. Some major extensions: the addition of a CYMK (cyan, yellow, magenta, and black) color model, which is essential for commercial printing processes; composite fonts that support very large character sets (the Japanese kanji, katakana, and hiragan alphabets, for example); and Display PostScript, which provides a device-independent imaging model for managing the appearance of monitors. The problem is that there is no way to tell whether clone manufacturers have updated their code in step with Adobe.

The recent announcement of PostScript, Level 2, promises to focus attention on the problem. PostScript, Level 2, incorporates all three of the extensions noted above, while adding several new commands as well. Level 2 products should hit the market early in 1991, and for at least the next year, both Level 1 and Level 2 products will continue to be introduced independently of one another. Adobe's assurance that any file written by a Level 1 driver will be imaged perfectly by a Level 2 interpreter doesn't guarantee the reverse. In some cases files written by Level 2 drivers will *not* be backward-compatible with Level 1 interpreters.

To this already heady mix of potential conflicts, you must add the uneven performance of PostScript drivers. Until the release of *Windows* 3.0, for example, Microsoft's own driver (released for *Windows* 2.*x* as an official driver for Adobe PostScript) printed any color information in a PostScript file as shades of gray. Not so good if you were connected to a color PostScript printer.

In most cases, however, PostScript files will continue to print without a hitch thanks to the extreme flexibility and stability of the language. The minitutorial that follows will help you get at least a feel for that language. It's not meant to turn you into a PostScript programmer overnight. But with some very basic understanding of PostScript, you can troubleshoot or at least identify many of the problems you may encounter. As you go through the examples presented here, you'll learn how to print .EPS files directly from DOS, fix incorrect or incomplete header information, rotate text for a photo credit, and proof color scanned images on early noncolor PostScript interpreters.

The PostScript Page Description Language uses vector-based instructions (rather than a bitmap) to express the way images will appear on a page. The instructions can relate to text, lines, objects, coordinates, and can even include scanned bitmaps. A PostScript program file can be stored in binary or ASCII format—the latter being the most common. In fact you can use a standard ASCII word processor to view, and even alter, the code in a PostScript file.

HANDS ON THE HEADER

Because you can bring it up as an ASCII file, the best way to start is by looking at a

real PostScript program, such as the Encapsulated PostScript file, which is shown in Figure 2a.

Encapsulated PostScript files are a subset of the PostScript language. .EPS files were developed so that PostScript images could be embedded into other documents—like a *PageMaker* newsletter.

There are only two basic differences between an .EPS file and the standard PostScript language files you can send to your printer: an .EPS file is limited to a single-page document; and just to be sure that all PostScript applications can import and print from them, .EPS files must be well-behaved.

A well-behaved .EPS file must adhere strictly to the PostScript structuring conventions of incorporating a header, a prologue, and a script. You'll recognize the header both because it's the first thing you encounter and because every line in it is preceded by a percent sign (%). This is a signal that these lines contain identifying information that should not be interpreted as part of the image.

Figure 2a demonstrates the type of information that should be included in the header, such as creation date and title. The importing application, such as *PageMaker,* looks for the percent sign followed by an exclamation point (%!). This identifies the file as Encapsulated PostScript code.

The importing application then uses the other information in the header to display the image on-screen. This sample display is extremely simple, consisting of a gray box, created from the Bounding Box comment, together with some tag lines picked up from the Creator and Title entries.

POSTSCRIPT, LEVEL 2

by Luisa Simone

In June of 1990, Adobe Systems announced a major upgrade to the PostScript page description language. In many ways, PostScript, Level 2, is more than Adobe's response to the needs of its customers and OEMs. It also represents a preemptive strike against Microsoft's as-yet-unreleased TrueImage technology. Some of the features in Level 2 will improve performance or increase the flexibility of the PostScript language. However, PostScript, Level 2, also attempts to bring new power to the PC desktop by establishing higher standards for imaging text and graphics.

PostScript, Level 2, introduces what may become yet another industry standard by providing a device-independent color model. Scanned full-color images—and the consequent need for color separations—are becoming commonplace in the PC graphics arena. The original PostScript had operators that understood RGB color only. The Adobe Color Extension specification brought the 32-bit CYMK model—a system essential to commercial printing processes—to the PC. Level 2 builds upon this by incorporating both RGB and CYMK systems into the basic code and by adopting an umbrella system—the CIE 1931 (XYZ) color space—that can be used to translate between the two.

The CIE (*Commission Internationale de L'Eclairage*) standard is based upon human perception itself. As such, it is a device-independent color model—unless you consider the human eye to be a device. If equipment manufacturers can find a way to calibrate the phosphors in monitors and the toner in printers to the CIE color model, we can have a color-matching system that will be consistent across all types and makes of hardware.

In addition, while the Composite Fonts extension made it possible to handle complex character sets (such as the 7,000-plus characters of the Japanese alphabets), PostScript, Level 2, makes it easier to issue font commands for all languages. The new `selectfont` operator can replace multiple operators like `findfont`, `scalefont`, and `setfont` that were necessary in Level 1 code.

Another highlight of Level 2 is its support of graphics compression routines such as CCITT, LZW, DCT, and JPEG. This should speed up performance when printing bitmapped photographic images—an important consideration if you pay for time on a Linotronic typesetter. Similarly, new combination operators such as `rectfill` (which draws a filled rectangle), should increase PostScript's speed, because the Level 2 interpreter can translate one command in place of what used to require a command sequence such as `moveto, lineto, lineto, lineto, closepath, fill`.

Type quality has also been improved by incorporating much of the font-rendering technology developed for Adobe Type Manager. You will still have to buy the ATM software product in order to see accurate on-screen representations of Type 1 fonts or to bring Type 1 fonts to your non-PostScript printer. However, Level 2 PostScript will give you cleaner, more-precise letterforms at small point sizes, and it will also reduce the time needed to render the specified point size from the outline font. Since speed—or the lack of it—has traditionally been the biggest complaint about PostScript's performance on text-intensive pages, this last item is good news for DTP aficionados.

As a matter of fact, the only bad news about Level 2 PostScript is that in some instances it may not be backward-compatible with Level 1 interpreters. Adobe admits that the migration to Level 2 among its OEMs may take up to 18 months. In this interim, some products that will be compatible with Level 1 and some with Level 2 will be coming to market.

Adobe's partial solution will be to develop smart PostScript drivers for the Mac, *Windows* 3.0, and Presentation Manager environments. Hopefully, these drivers will create an either/or scenario. That's to say, either the new, more-efficient operators will be executed or the prologue will redefine the new command using older, definitely compatible Level 1 operators.

The original philosophy behind the PostScript language was to sacrifice speed, if necessary, in favor of functionality. Level 2 PostScript promises to give users both kinds of high performance—a high-quality printout and record time. ■

CREDIT.EPS and .PS | **COMPLETE LISTING**

```
Version (a): CREDIT.EPS

%!PS-Adobe EPSF
%%Title: Rotated Credit
%%Creator: GK
%%CreationDate: 9/27/1990
%%BoundingBox: -10 0 2 500
/r {rotate} def

0 setgray
/Helvetica findfont 5 scalefont setfont
0 0 moveto
90 r
(ILLUSTRATION: LUISA SIMONE) show
-90 r
```

A

```
Version (b) CREDIT.PS

%!PS-Adobe EPSF
%%Title: Rotated Credit
%%Creator: GK
%%CreationDate: 9/27/1990
%%BoundingBox: -10 0 2 500
/r {rotate} def
/inch {72 mul} def

1 inch 1 inch translate

0 setgray
/Helvetica findfont 5 scalefont setfont
0 0 moveto
90 r
(ILLUSTRATION: LUISA SIMONE) show
-90 r

showpage
```

B

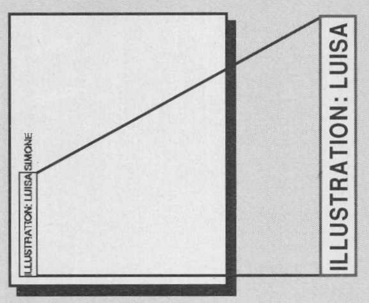


Figure 2: The 0,0 coordinates in version (a) place the output within the nonprinting border of the HP laser engine, thus the output is not visible. By using your word processor to make the slight ...difications that turn version (a) into version (b) (where the output is visible), you can print an .EPS ... on your PostScript printer with the DOS COPY command. (The output shown in this figure has been reduced for reasons of space.)

Think of the gray box—which is also called a `bounding box`—as a placeholder. The placeholder is necessary because Display PostScript doesn't exist on the PC platform. So there is no way for an importing application, such as *PageMaker,* to display a plain-vanilla .EPS file on-screen. The gray box is created from a Bounding Box comment in the file's header and tells the importing application the vertical and horizontal dimensions (or boundaries) of the .EPS file.

If for some reason the bounding box information is wrong, the EPS image will be clipped when it prints. Why, then, is it handled only as a comment? The answer is that by keeping the bounding box information as a comment, the importing application can stack up sizing commands in front of the .EPS file. That allows you to resize the image without loss of quality.

I tend to keep a lot of .EPS files, culled from different sources, on hand for testing and design. Customizing the header infor-...on is quite a simple matter. In the ex-a..ple shown here, the title of the file is "Rotated Credit," since that is what this program does. It prints a photo or illustration credit vertically—something, incidentally, that *PageMaker* 3.01 still can't do.

You might decide to title an .EPS file "Company Logo." You'll want to change the creator name (which is usually something totally useless, like Lotus Freelance) to the name of the person or department in your organization responsible for the data. In this example, the GK stands for Gerard Kunkel, *PC Magazine*'s director of design and electronic publishing.

A word of warning: if you ever pull an .EPS file into your word processor and the %! identifier is *not* the first thing you see, you have probably opened an .EPS file with an attached TIFF header.

Think of an attached TIFF image as a kind of courtesy. Some programs attach this low-resolution bitmapped image onto an .EPS file so that you can see something more than just a gray bounding box on the screen. TIFF is a binary file format that does not get along well with ASCII-based word processors. Your options are either to abort the file, or—if you are determined to work with the PostScript code anyway—to delete any binary information that you see.

THE IMPORTANCE OF PROLOGUES

Though it's not absolutely necessary, having a *prologue* will in most cases improve the efficiency of a PostScript file. The prologue routines are identified by a slash (/). Basically, they are used to create new definitions that can, for example, combine a series of simple but related commands into one procedure. The power inherent in these on-the-fly definitions increases exponentially when you consider that you can include variables, employ looped expressions, and store the new routines in user-defined dictionaries.

Though the program in Figure 2a is really too simple to require a prologue, I included one anyway. Here I redefined the `rotate` command as `r`. Now whenever I type the newly defined `r` command, a rotation will occur.

Prologues can also be used by smart programmers—and smart drivers—to overcome some of the barriers between different versions of PostScript. Suppose, for example, that a PostScript Level 2 driver wanted to use one of the new combination commands, such as `rectfill`. This single command draws a filled rectangle and obviates the need to list six separate line and fill commands.

A Level 1 interpreter will not understand the `rectfill` instruction, of course, so it won't be able to print the file. A well-written prologue can save the day, however, by instructing the program to look for the command in the interpreter. If the command is not present, the prologue will simply redefine the `rectfill` instruction as the very series of simpler commands known to be part of Level 1 PostScript. In fact, Adobe is planning to write just such a smart driver for the Mac environment, as well as for *Windows* 3.0 and Presentation Manager.

READY, SET, SCRIPT

The actual commands are listed in the portion of the program known as the *script*. Commands built into the PostScript language are called *operators*. Even if you know nothing about the PostScript language, you'll recognize a fair number of its operators. Obviously, `moveto` is a command that goes to a specified *x*- and *y*-coordinate. `lineto`, `scalefont`, `findfont`, and `curveto` have similarly self-evident meanings.

The actual script of our sample .EPS file begins with a `setgray` command that establishes a zero (or black) value. The commands then simply run in a logical se-

quence: find a font named Helvetica, scale it to 5 points, and set it. Then move to the lower-left corner of the bounding box (0,0), rotate 90 degrees, and show the text string that appears within the parentheses.

To use this file yourself, simply replace my name with a single line of text. Be sure to leave the parentheses intact, however: that is how the PostScript interpreter identifies text strings. The bounding box has been made arbitrarily tall so that most names will fit without changing the header. However, in these days of hyphenated names, you may find that the last few letters of your copy get clipped when the file prints. If this happens, you can increase the size of the bounding box.

The four numbers that follow the bounding-box comment determine its size. The first two numbers set the *x*- and *y*-coordinates of the lower-left corner of the box, the second set of numbers set the *x*- and *y*-coordinates for the upper-right corner of the box. Obviously, the difference between them determines the size of the bounding box. So when the bounding box comment is

```
%%BoundingBox: -10 0 2 500
```

the width will be 12 points (more than sufficient for the 5-point type we are setting) and the height will be 500 points—good enough for most names.

Changing the last value from 500 to 600 or more should solve any problems. Once the file is saved to disk, you can pull it up in *PageMaker* and use it just like any other .EPS file.

If you have been following the logic of this little .EPS file closely, you'll have noticed that numbers and text strings always precede the commands. That's because PostScript uses postfix notation (often called *Reverse Polish Notation*), in which the data is always listed first, followed by the operators that will act upon that data. Thus, the `moveto` operator is preceded by the appropriate *x*- and *y*-coordinates.

Postfix notation is necessary because of the way PostScript stacks all operations in memory. For example, if you were to issue an `add` command, PostScript would pop the two topmost numbers off the stack, perform the calculation, and then push the result back onto the stack for further manipulation. Without the correct number of elements in the stack, all you'll get at printout time is an error message.

The best-behavior rule applies not only to the structure of an .EPS document, but to its operators as well. Drawing operators are considered well-behaved; an `erase-page` command is not. Since Encapsulated PostScript images are designed to be transported between applications, it would be foolish to allow the use of operators that could destroy data embedded within another PostScript program. A dangerous operator such as `initgraphics`, which reinitializes the graphics environment, could wipe out not only the EPS image, but the entire page of a *PageMaker* document on which it appears.

In addition to standard and .EPS files, there is a third PostScript format: *Adobe Illustrator* files. PC users are growing increasingly familiar with the .AI extension and with the benefits provided by what is often referred to as editable EPS. *Illustrator* files take the good behavior rules to the *n*th degree. Indeed, this format is so restricted that all of the operators can be defined in the prologue.

For example *Adobe Illustrator* 1.1—the most often used format for compatibility's sake—has such a limited set of operators that it uses four connected arcs to describe a circle, rather than having a separate circle operator.

Since the information is so easy to parse, many graphics applications have taken to translating PostScript operators into their own native format. In effect, such applications are implementing partial PostScript interpreters. Once the PostScript file is translated, the image can be edited.

Regardless of which format you work with, .PS, .EPS, and .AI files are all true PostScript programs, so they can be sent to a PostScript device for interpretation. In fact, with just a little tweaking, you can even use the DOS COPY command to proof .EPS files directly from your hard disk to a PostScript printer.

PROOFING .EPS FILES FROM DOS

If all you need is a quick printout of an .EPS file, there's absolutely no need to load up *PageMaker*, create a new document, import the file, and then print out to your PostScript device. The modified .EPS file shown in Figure 2b can be printed directly from DOS.

The easy part is adding the `showpage` command to the very end of the file. Without this, the interpreter will create the image in memory, but will never apply toner to the page. Normally, .EPS files rel upon the importing application to issue the `showpage` command. Don't worry about the extra carriage returns (blank lines):

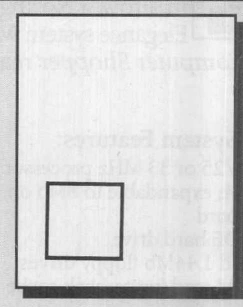
COMMENT.PS — **COMPLETE LISTING**

```
Version (a)

%!PS-Adobe EPSF
%%Title: Uses of %
%%Creator: LS
%%CreationDate: 11/30/1990
%%Boundingbox 0 0 300 300
%%EndProlog
100 100 moveto
300 100 lineto
300 300 lineto
100 300 lineto
closepath
10 setlinewidth
stroke
showpage


Version (b)
%!PS-Adobe EPSF
%%Title: Uses of %
%%Creator: LS
%%CreationDate: 11/30/1990
%%Boundingbox 0 0 300 300
%%EndProlog
100 100 moveto
300 100 lineto
300 300 lineto
%100 300 lineto
closepath
10 setlinewidth
stroke
showpage
```

A



B

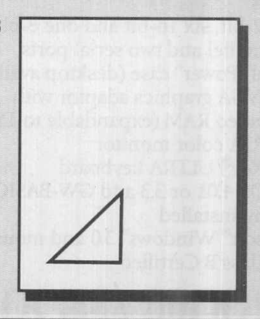


Figure 3: By commenting out the `100 300 lineto` command with a %, PostScript will turn the square created by version (a) into a triangle (version b). (The output shown in this figure has been reduced for reasons of space.)

PostScript will continue to interpret a file as long as there is a stream of information coming down the pike. I leave the carriage returns in to segregate the temporary operators I've inserted for printing—it makes it easier to delete them after I've proofed the file.

At this point, you've told PostScript to print the image. The only problem is that you haven't told PostScript *where* to print the image.

Remember the bounding box information is only a comment in the header. And as a comment, only importing applications (such as *PageMaker*) can use this information, not a PostScript interpreter. These applications use the coordinates in the bounding box to size the .EPS file when it imports it. But since you are printing it directly from DOS, you must tell the interpreter the positioning information that the target application would normally supply. And you'll have to insert this before you can actually send this file to a printer.

At the moment, the only positioning instruction is a `0 0 moveto` command. This will print the photo credit at the extreme lower-left corner of an 8.5- by 11-inch page. PostScript assumes an *x*- and *y*-coordinate system that covers the entire page. The grid is set up in points (points and picas are the measurements commonly used in commercial design and printing environments). Given a standard letter-size sheet of paper, with coordinates starting at the lower-left corner (0,0), the furthest point (the upper-right corner) is designated at 612 (for the *x*-coordinate), and 792 (for the *y*-coordinate). If you do some quick math, you'll see that there are 72 points per inch.

One special consideration applies to those of you working with PostScript-enhanced LaserJets. PostScript will let you position an image anywhere on its grid: there are margins built into the hardware. The 0,0 coordinates listed in the .EPS file of Figure 2a can thus be imaged by PostScript, but those coordinates fall within nonprinting border of the HP laser engine. If you position your image at 0,0—there won't be an interpreter error, but the engine won't be able to print at least some portion of the file.

Just to be safe, then, I've built a temporary *x,y* location into Figure 2b. This location also provides a good example of prologue definitions in action. The line

```
/inch {72 mul} def
```

in the prologue means that all subsequent move commands using the word `inch` will be multiplied by 72 points. The line

```
1 inch 1 inch translate
```

in the script portion "translates" to a point 1 inch from both the left and bottom edges of the paper into my new point of origin for the credit line. (You might find it helpful to distinguish, as I have done, between the original Encapsulated PostScript file of Figure 2a and the modified, printable version of Figure 2b, by saving the latter with a .PS rather than the original .EPS extension.) Everything is now in place to print this file directly from the DOS prompt by issuing the command

> **PostScript still has the decided edge in typeface manipulation. It can rotate, pattern fill, perform vector clipping, and distort letterforms while still in text mode.**

```
Copy Credit.ps lpt2:
```

The basic skills you've just learned can be valuable in troubleshooting some of the more common problems with .EPS files. If a faulty driver were to write out a too small bounding box, it is a simple matter to fix. Just print the .EPS file out to the printer, using the methods discussed here. Then measure the live area of the picture, convert the dimension information to points, and type in an appropriate bounding box comment.

Another trick you may wish to employ in troubleshooting is to use the percent sign (%) to turn a command into a comment. Remember, PostScript will not attempt to interpret anything preceded by a percent sign. Temporarily commenting out questionable lines of code has often allowed me to import and print a problem .EPS file. True, this technique works best on small files, and it helps if you can easily identify the potential problem. A `moveto` command with only one coordinate position is enough to bomb out the interpreter, for example. But I have actually had .EPS files that printed out perfectly with a few extra percent signs added to the file.

To see the effect of a single line of PostScript code, consider Figures 3a and 3b. In the (a) version of this program, PostScript draws a square. The (b) version comments out the third `lineto` operator but continues with the `closepath` command. PostScript responds by drawing a triangle.

Someone once told me that the EPS format is like a black box—you can't see what's inside. The examples I've presented prove that you can see what is inside the file. However, when something does go wrong, it is often difficult to locate the exact command that's causing the error. The PostScript interpreter doesn't print out partial files, and herein lies the difficulty. If an operator blows up even one line before `showpage`, nothing will print.

For this reason, if you're faced with a recurring need to debug .EPS files, or if you want to pursue PostScript programming seriously, you should consider getting a copy of EHANDLER.PS from Adobe Systems. This little utility will print out whatever was in memory before the error occurred—allowing you to more easily locate the offending operator.

TEXT MANIPULATIONS

For most business users, PostScript's most important benefit may be the Adobe Type 1 scalable font technology. Until the introduction of AGFA Compugraphic's Intellifont Scalable Typefaces, Adobe's outline font technology was the *only* way you could resize typefaces on the fly without a loss of quality. However, the introduction of Intellifont typefaces, together with Hewlett-Packard's support for them in PCL 5, have given users a scalable alternative to Adobe Type 1 fonts.

HP is clearly targeting the business-user market with PCL 5 and the Intellifont technology. Indeed, with a range of 0.25 to 999.75 points (in quarter-point increments), eight resident scalable fonts (weights each of CG Times and CG Univers), and the ability (at long last) to print in either portrait or landscape mode, PCL 5

MATRIX1 and MATRIX2.PS — **COMPLETE LISTING**

```
(a) Matrix1.PS

%!PS-Adobe EPSF
%%Title: Condensed Matrix
%%Creator: LS
%%CreationDate: 11/30/1990
%%Boundingbox 0 0 110 72
%%EndProlog

%Condensed type
0 setgray
/Helvetica-Bold findfont [23.7 0 0 72 0 0] makefont setfont
100 100 translate
0 0 moveto
(MATRIX) show
showpage
```

A


```
(b) Matrix2.PS

%!PS-Adobe EPSF
%%Title: Oblique Matrix
%%Creator: LS
%%CreationDate: 11/30/1990
%%Boundingbox 0 0 290 72
%%EndProlog

%/Reverse Oblique
0 setgray
/Helvetica-Bold findfont [64.8 0 -45 72 0 0] makefont setfont
100 100 translate
0 0 moveto
(MATRIX) show
showpage
```

B

Figure 4: The bracketed numbers preceding the `makefont` command change the appearance of the font drastically. A detailed explanation of their meaning is given in Figure 5. (The output shown in this figure has been reduced for reasons of space.)

may provide all the font technology that office users need. And since many of the remaining restrictions—if you can call them that—are imposed by the printing engine, there is great potential for Intellifont Scalable Typefaces to migrate to higher-resolution devices.

At the moment, however, PostScript still offers advantages for the professional designer and the truly finicky desktop publisher. Compare the eight resident fonts of the HP LaserJet III with the 13 or 35 resident typefaces in a PostScript printer. With over 6,000 designs from Adobe and major type vendors like Monotype, there are simply more Type 1 fonts available. Because PostScript formats are independent of the hardware, Type 1 fonts are guaranteed to be stored in identical form on desktop laser printers and on Varityper, Linotronic, and even AGFA Compugraphic high-resolution typesetters. This ensures that the proof you get is an accurate preview of the final typeset output.

When it comes to manipulating typefaces PostScript still has a decided edge over PCL 5. Though the Hewlett-Packard PDL can create many special effects with letterforms, it can only do so *after* it has converted the text to a graphical vector outline. By contrast, PostScript can rotate, pattern fill, perform vector clipping, and distort letterforms while still in text mode.

The two small PostScript files shown in Figure 4 illustrate how a single PostScript operator, `makefont`, can be used to manipulate and distort a six-value font matrix. The numerical values that appear in brackets control the general appearance of a Type 1 font and are diagrammed in Figure 5. Specifically, the first number scales the width of the letterform. The second number changes the sine value. The third number is the negative sine value and can create oblique transformations such as italics. The fourth number scales the height of the letter. The fifth number affects horizontal positioning and moves the text along the *x*-axis. The sixth number affects vertical positioning and moves the text along the *y*-axis.

Merely by changing one value in the matrix, you can dramatically alter the appearance of the typeface. In the first instance, I've created a condensed typeface by entering a width value one-third of the height value. In the second version I've created a reverse oblique effect by using a negative value for the negative sine (a positive value would have resulted in a standard italic face).

Just for a moment, consider how easy it is for you to understand these two small programs. PostScript designates fonts by name and the `findfont` operator. Thus, if you need to change the typeface at printout time and you just don't have time to recompose a long PostScript document, you can make the necessary changes with nothing more than your word processor. Changing the font used in these PostScript files is as simple as replacing the word Helvetica-Bold with Times-Roman.

Although this brief look at PostScript has explored neither the depth nor the elegance of the language, it has given you a feel for the language and a number of basic troubleshooting skills. If you are intrigued by what you've seen here, you can get further information in *PostScript Language Tutorial and Cookbook* and *PostScript Language Reference,* both written by Adobe Systems and published by Addi-

HOW POSTSCRIPT SHAPES FONTS

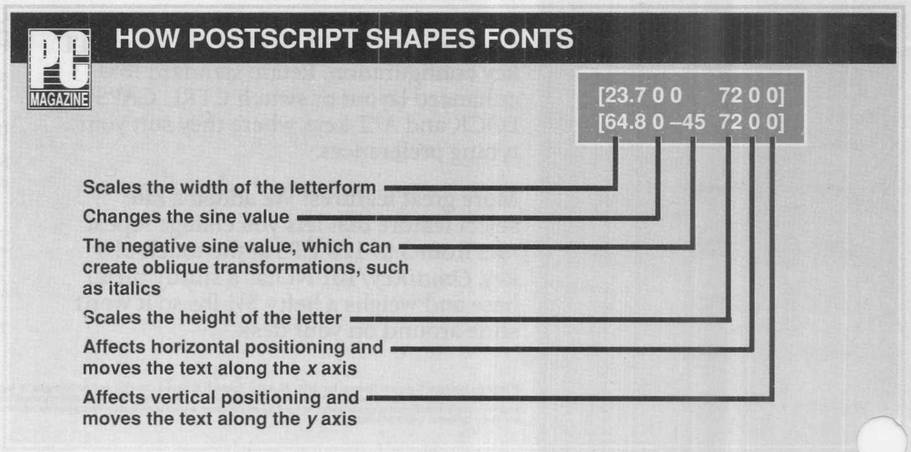


Figure 5: By simply changing the bracketed numbers in a single command, PostScript turns out two very different typefaces from the same font.

son-Wesley Publishing Co. (New York, 1985). In addition, *Graphic Design with PostScript*, by Gerard Kunkel (Scott, Foresman and Company, Glenview, Illinois, 1990) will guide you from the most elementary concepts through sophisticated programs.

Of course, PCL 5 and the AGFA Compugraphic Intellifont technology will continue to improve. Moreover, Microsoft has announced—but still not yet released—TrueImage and TrueType, which also promise to deliver much of the functionality of PostScript (including Display PostScript) as part of the *Windows* and Presentation Manager environments. How all of this fits together with PostScript Level 2 and the Adobe Type Manager (which brings Display PostScript technol-

PostScript has clearly changed the face of PC-based electronic publishing design.

ogy to the *Windows* 3.0 environment, though only for text) remains to be seen.

From a user's perspective, however, it is reassuring to see that the two current major players are seeking peaceful coexistence. Hewlett-Packard is now offering an official Adobe PostScript Cartridge for the LaserJet III ($695). And Adobe has just introduced a $495 PostScript Cartridge for the LaserJet Series II.

PostScript's history is clear. John Warnock and Charles Geschke's seminal work at Xerox PARC (Palo Alto Research Center) and the subsequent development of PostScript have changed the face of PC-based electronic publishing design. If the future of PostScript is unpredictable, it is certainly exciting. And this basic introduction to the paradigm of page description languages should help you get the most from PostScript's current and future capabilities.

Luisa Simone is a contributing editor of PC Magazine. ■